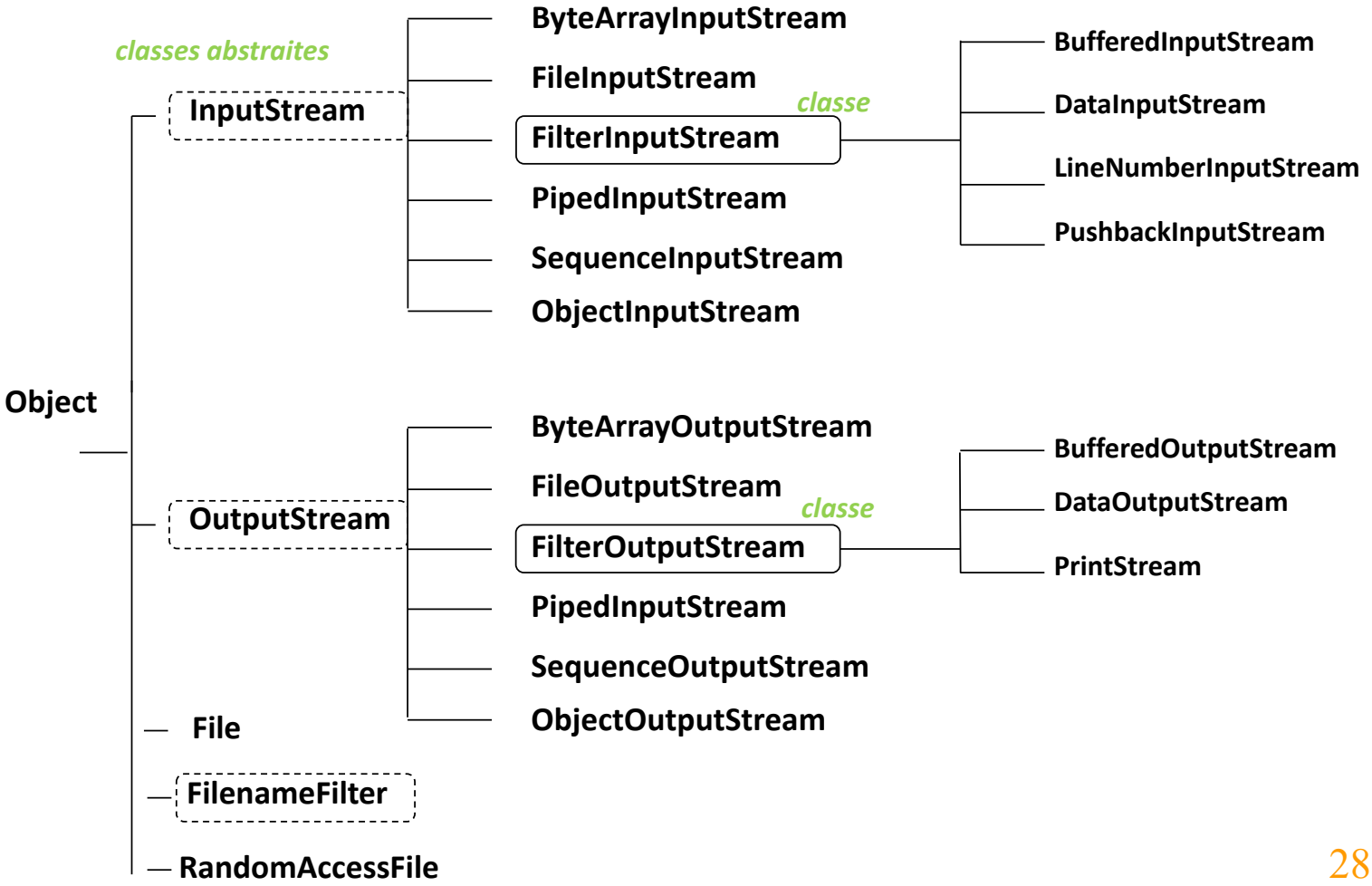
Etape 2

- Flux binaire en Lecture / Ecriture dans un fichier binaire

Famille des flux binaires

Les classes de l'API des flux se situent dans le paquetage :

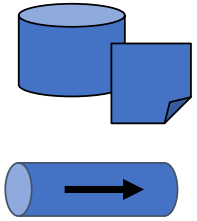
- ***java.io.**** pour les autres opérations d'entrées/sorties.



1.2 Manipulation des flux

Il faut distinguer :

- Le « stockage » des données : fichiers, BD, Internet : sources de données ou destinations de données ;
- Transmission d'informations : flux = vecteurs de communications, qui servent à lire ou écrire dans les lieux de stockage.



Certains flux sont spécialisés à une source/destination de données :

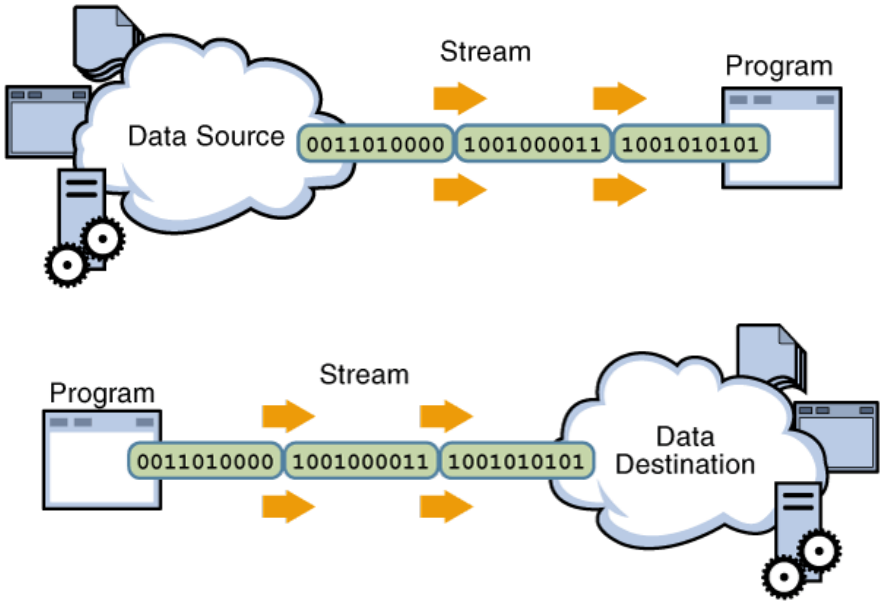
- Flux basés sur les fichiers, chaînes de caractères, tableaux, buffers ;
- Flux basés sur les sockets réseaux ;
- Basés sur les consoles d'applications ;

mais leur utilisation est similaire.

1.2 Manipulation des flux

Flux (Stream) : canal de communication unidirectionnel parcouru par les données

- Flux d'entrée : envoie de données d'une source à un programme ;
- Flux de sortie : envoie de données d'un programme à une destination ;



Classes de flux de données

Rapides et efficaces mais pas compréhensibles par le user

Par exemple, l'entier (int) 1234 correspond à la séquence de 4 octets : 00 00 04 D2

Deux principaux :

- Les flux d'octets "{Input / Ouput} Stream" : transportent des byte, octets, entiers compris entre 0 et 255 (qui peuvent quasiment tout représenter) ;
- Les flux caractères "Reader/Writer" : flux d'octets spécialisés qui ne traitent que des données textuelles : Unicode (2Octets) .

moins efficaces mais lisibles par le user

	Flux d'entrées	Flux de sorties
JDK 1.0 Flux d'octets (8 bits)	*InputStream* • *FileInputStream* • *DataInputStream* • *BufferedInputStream* • …	*OutputStream* • *FileOutputStream* • *DataOutputStream* • *BufferedOutputStream* • …
JDK 1.1+ Flux de caractères (16 bits)	*Reader* • *FileReader* • *BufferedReader* • *StringReader* • …	*Writer* • *FileWriter* • *BufferedWriter* • *StringWriter* • …

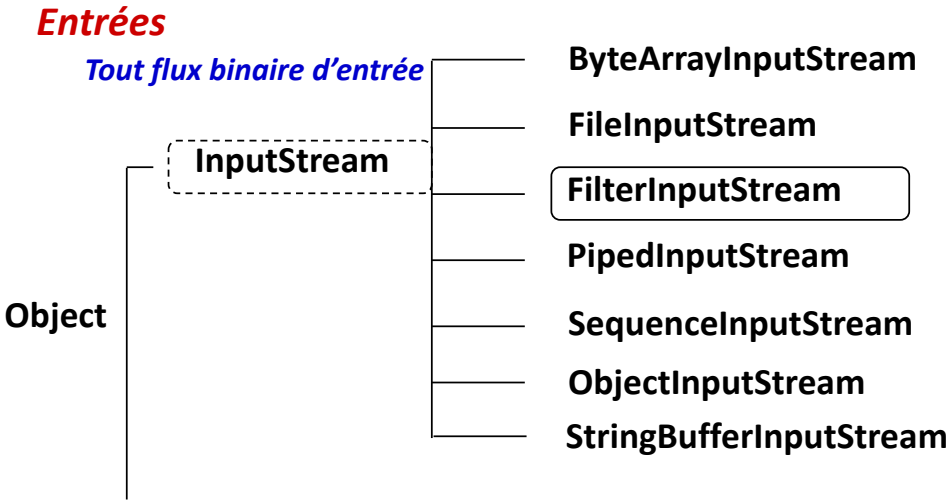
Deux classes permettent de faire la transition des « streams » vers des « readers/writers » :

- ***InputStreamReader*** permet de transformer un ***InputStream*** en un ***Reader***,
- ***OutputStreamWriter*** permet de transformer un ***OutputStream*** en un ***Writer***.

flux binaires en lecture

- **Flux binaire d'entrée ou de lecture**

  - Héritent de la classe InputStream (abstraite) et fournissent des méthodes qui permettent la lecture séquentielle de données : *read* ;



Comportement générique de la méthode *read()* :

int read() retourne :

- un int qui contiendra l'octet lu à la position courante dans le flux (le curseur se déplace à l'octet suivant) (seul l'octet de poids faible est considéré)
- -1 si l'on a atteint la fin du flux ;

la classe abstraite InputStream

- ***int read(byte[] b)*** : permet de lire plus d'un octet à la fois. Les octets lus sont stockés dans le tableau passé en paramètre. La méthode lit autant d'octets qu'elle peut en stocker dans le tableau et autant qu'elle peut en lire sur le flux. La méthode retourne le nombre réel d'octets lus et -1 si la lecture a échoué (fin du flux).

- ***int read(byte[] b, int off, int len)*** : similaire à la méthode précédente. Au lieu de remplir le tableau dès la première position, on commence à écrire les octets à la position *off (offset)* du tableau. Le paramètre *len* correspond au nombre maximum d'octets à lire.
  l'entier retourné est le nombre réel d'octets lus et -1 si la lecture a échoué (fin du flux).

- ***int available()*** : retourne le nombre d'octets qui peuvent être lus (ou passés) sur le flux d'entrée sans bloquer la prochaine lecture.
- ***close()*** : ferme le flux et libère toutes les ressources systèmes liées à ce flux. Cette méthode doit toujours être appelée une fois que vous avez fini de travailler avec le flux.

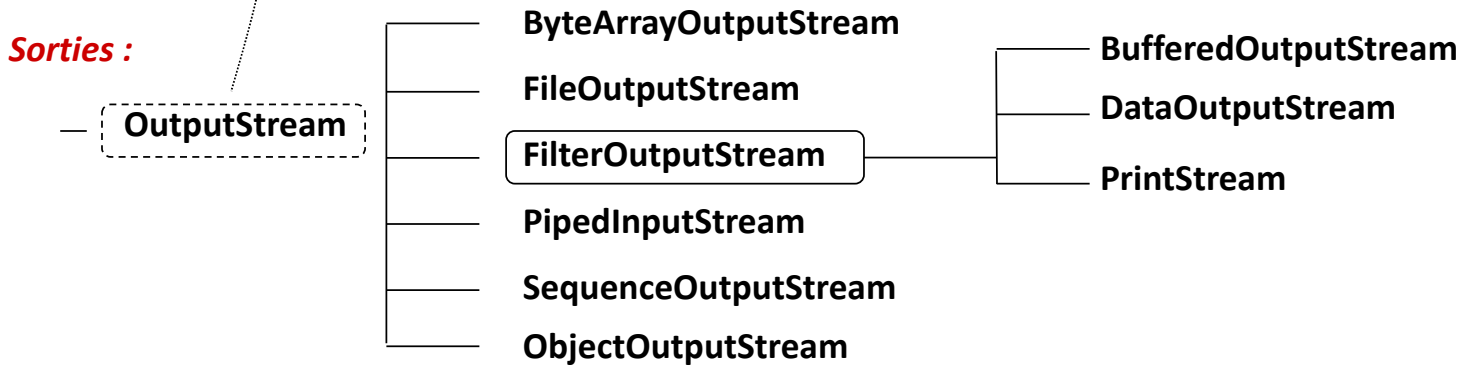
- ***mark(int readlimit)*** : marque la position courante dans le flux. Le prochain appel de la méthode reset() repositionne le flux à cette position, ainsi la prochaine lecture relira les octets à partir de la marque posée. Le paramètre *readlimit* spécifie le nombre d'octets pouvant être lus. Lorsque cette limite est atteinte la marque disparaît.

- ***reset()*** : repositionne le flux à la dernière position définie par la méthode mark().
- ***boolean markSupported()*** : teste si le flux supporte les méthodes mark() et reset().
- ***skip(long n)*** : permet de passer et d'ignorer n octets de données sur le flux.

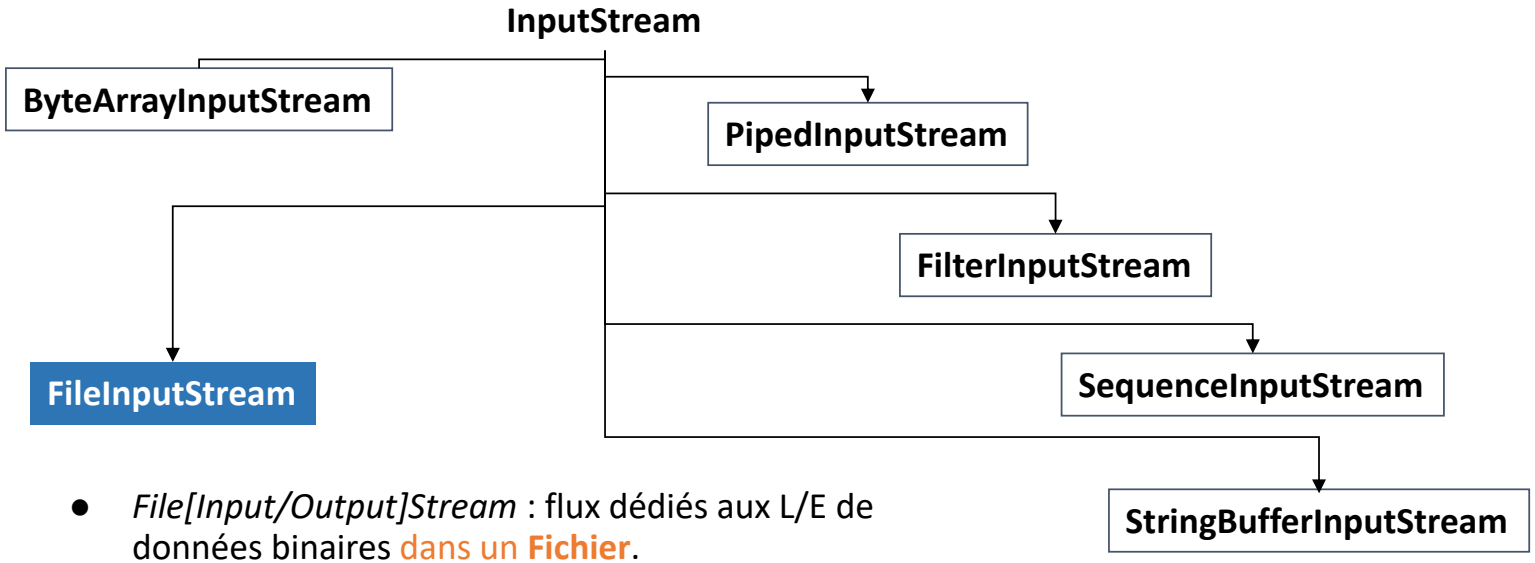
Les flux binaires en écriture

- **Flux binaire d'écriture (sortie)**

void write(int) écrivent l'octet courant sur le flux (le curseur se déplace à l'octet suivant);
(seul l'octet de poids faible est considéré)



FileInputStream



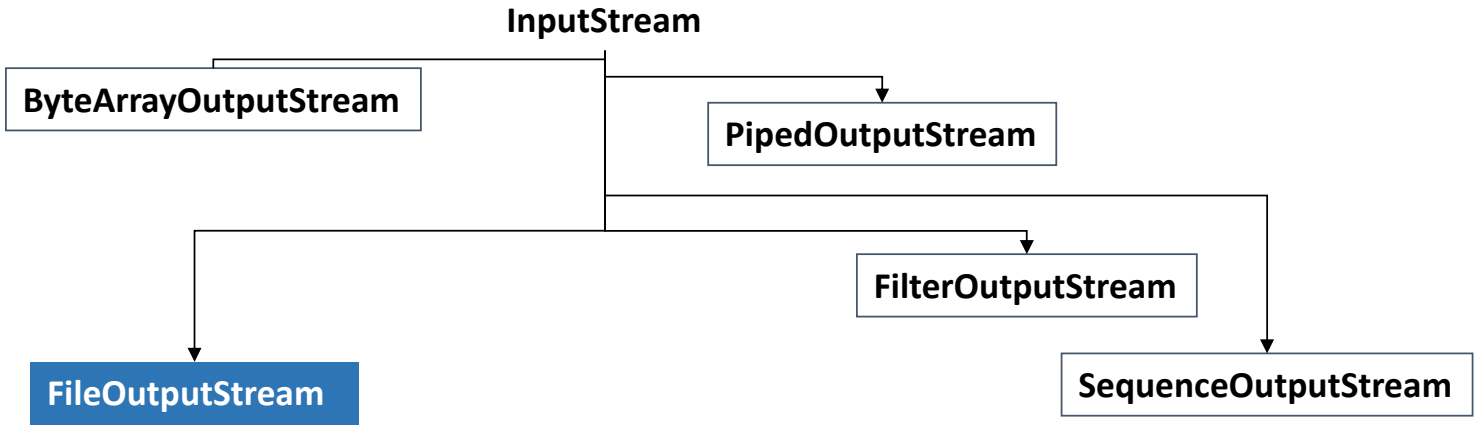
- *File[Input/Output]Stream* : flux dédiés aux L/E de données binaires dans un **Fichier**.
  - *(par exemple, pour lire une image)*

- *Constructeurs* :

FileInputStream(File file) : création d'un *FileInputStream* en ouvrant une connexion avec un fichier **existant** dont le nom est spécifié dans l'objet File (*FileNotFound* si le fichier n'est pas trouvé mais il n'est pas créé)

FileInputStream(String name) : idem mais en donnant directement le nom externe du fichier

FileOutputStream

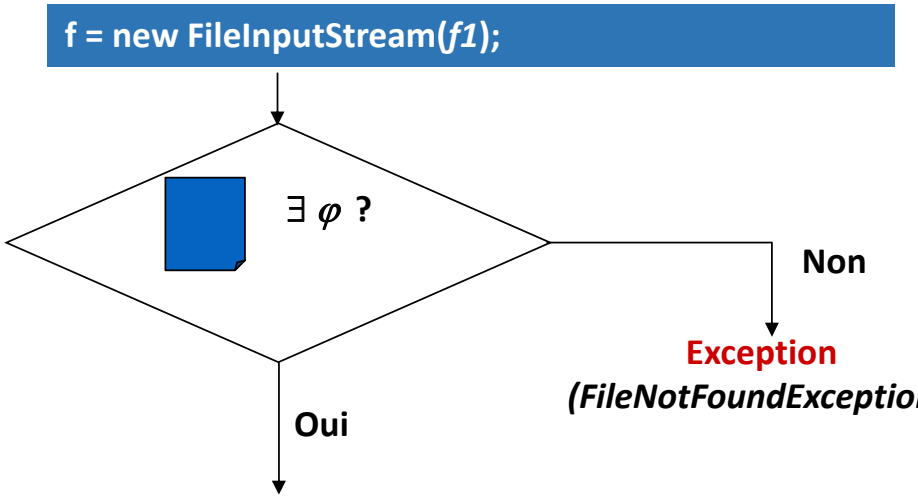


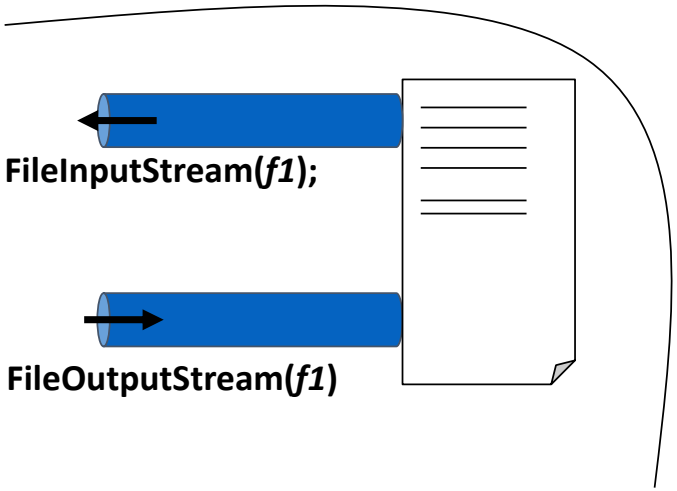
- *Constructeurs (entre autres)* :

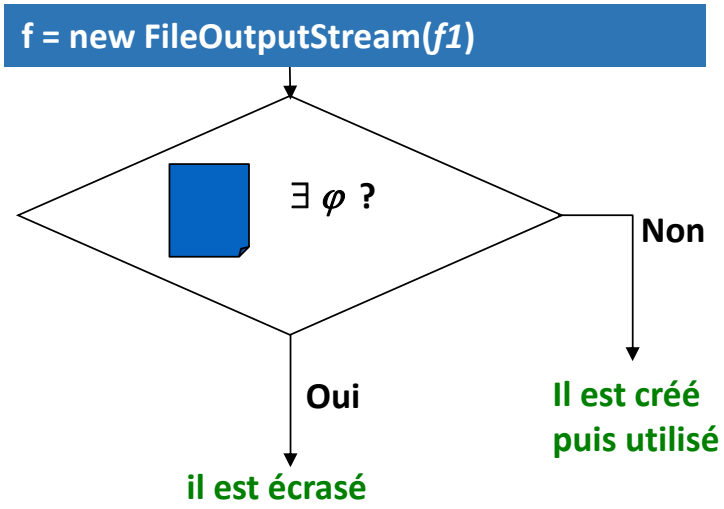
FileOutputStream(File file) : création d'un flux en écriture connecté à un fichier **existant** dont le nom est spécifié dans l'objet File (le fichier physique est CREE s'il n'est PAS TROUVE)

FileOutputStream(String name, boolean append) : idem, si *append* est vrai, le fichier est ouvert en ajout sinon on réécrit du début.

Comportement du File/Input/OutputStream







Flux en entrée : principe d'utilisation

- Algorithme de traitement par flux en Lecture :

0. => déterminer quels flux on va utiliser en fonction de la source et de ses besoins
1. s'assurer que la source est disponible
2. créer le flux en lecture en traitant les éventuelles exceptions (le curseur se place en début de la source)
3. **Tant Que** l'on a des octets à lire dans la source
    1. on lit une quantité d'octets (le curseur va avancer d'autant)
    2. gérer les exceptions en lecture
    3. faire d'éventuels traitements avec les octets lus
4. **Fin Tant QUE**
5. fermer le flux

Flux en sortie : principe d'utilisation

- Algorithme de traitement par flux en Ecriture :

0. => déterminer quels flux en écriture on va utiliser en fonction du type de destination
1. **s'il** s'agit d'un nouveau stockage s'assurer que la destination n'existe pas déjà
2. **sinon** s'assurer que la destination est disponible (on écrira dedans)
3. créer le flux en écriture en traitant les éventuelles exceptions (par défaut le curseur se place en début de destination)
4. **Tant Que** l'on a des octets à écrier dans la destination
    1. on écrit une quantité d'octets (le curseur va avancer d'autant)
    2. gérer les exceptions en écriture
5. **Fin Tant QUE**
6. Fermer le(s) flux

Flux en entrée : exemple d'utilisation

```java
public int compteOctets(String nomFichier) {

    File f = new File(nomFichier);
    int r;
    int cpt = 0;

    if (!f.exists() || f.isDirectory()) {
        System.err.println("Fichier inexistant ou présent comme dossier");
        System.exit(-1);
    }

    try (FileInputStream fis = new FileInputStream(f)) {
        do {
            r = fis.read();
            if (r != -1) cpt++;
        } while (r != -1);

    fis.close();   // pas indispensable

    }
    catch (FileNotFoundException e) {e.printStackTrace();}
    catch (IOException e1) {e1.printStackTrace();}

    return cpt;
}
```

Flux en Sortie : exemple d'utilisation

```
public void copieOctetParOctet(String nomFichier) {
    try {
        FileInputStream fis = new FileInputStream(nomFichier);
        FileOutputStream fos = new FileOutputStream("SousRep" +
        File.separator + nomFichier);

        while (fis.available() != 0) {
            int o = fis.read();
            fos.write(o);
        }
        fos.close();
        fis.close();
        }
        catch (FileNotFoundException e) {e.printStackTrace(); }

}
```